Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andrej Česen

# Algoritmi za čiščenje pomnilnika

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2014



*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pravilno upravljanje s pomnilnikom je ena od pomembnejših programerskih nalog. Posledica napačnega upravljanja je lahko počasno delovanje programa (in morebiti celo sistema), pogosto pa tudi nepredvidljivo delovanje, kar privede do predčasnega končanja izvajanja programa ali napačnih rezultatov. Glavni del zgodbe o upravljanju pomnilnika predstavlja sproščanje delov pomnilnika, ki niso več v uporabi. Moderni programski jeziki, za razbremenitev programerja eksplicitnega upravljanja in za preprečitev posledičnih napak, uvajajo koncept avtomatskega čiščenja pomnilnika.

V diplomskem delu preglejte področje čiščenja pomnilnika in predstavite različne pristope k reševanju tega problema. Natančneje opišite algoritme označi-počisti, označi-strni, kopiranje in štetje referenc. Implementirajte čistilca pomnilnika in primerjajte izvorno kodo programa brez uporabe in z uporabo čistilca.

# Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Česen, z vpisno številko **63100177**, sem avtor diplomskega dela z naslovom:

*Algoritmi za čiščenje pomnilnika*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2014 Podpis avtorja:

*Na tem mestu bi se rad zahvalil mentorju doc. dr. Tomažu Dobravcu za vodenje pri izdelavi in za strokovni pregled diplomskega dela. Hvala tudi vsem, ki ste mi pomagali pri nastajanju diplomskega dela. Posebno zahvalo namenjam družini, ki mi je ves čas študija stala ob strani.*

# Kazalo

# Seznam uporabljenih kratic

| kratica | angleško | slovensko |
|---|---|---|
| **ABI** | application binary interface | aplikacijski strojni vmesnik |
| **API** | application programming interface | programski vmesnik |
| **FIFO** | first in first out | prvi dodan element obdelan prvi |
| **LIFO** | last in first out | zadnji dodan element obdelan prvi |
| **ZCT** | zero count table | tabela objektov z ničelnimi števci |

# Povzetek

Diplomsko delo je posvečeno implementaciji avtomatskega upravljanja pomnilnika za programski jezik C. Metoda čiščenja označi-počisti se je prilagodila za nekooperativen jezik, ki ne sporoča informacij o tipih podatkov na mestih dostopnih mutatorju. Prepoznavanje kazalcev v korenih in poljih objektov je zato konzervativno, kar omogoča varno delovanje čistilca - če je vsebovana vrednost dovolj podobna kazalcu, se jo smatra kot kazalec (četudi to morda ni). Da se uporabnikove podatke zavaruje pred nenamernim pisanjem čistilca, se je označevalne bite premaknilo iz glav objektov v bitna polja, shranjena v ločenem delu pomnilnika. Na koncu sledi še vrednotenje uporabe čistilca v praksi.

**Ključne besede:** čiščenje pomnilnika, označi-počisti, označi-strni, prepisovanje, štetje referenc.

# Abstract

This thesis focuses on an implementation of automatic memory management in C programming language. Mark-sweep method was modified for use in uncooperative programming language, which does not share data type information of memory slots accessible by the mutator. Due to this fact, decisions on pointer identity are conservative which guarantees safe collector operation - if value looks sufficiently like a pointer, it is considered a pointer (although it might not actually be one). Mark bits were moved from object's headers to bitmaps, stored in a seperate part of memory to prevent accidental writes to user's data by the collector. Finally, the usage of garbage collector was evaluated in practice.

**Keywords:** garbage collection, mark-sweep, mark-compact, copying, reference counting.

# Poglavje 1

# Uvod

Pomanjkanje pomnilnika kot vira računalniških sistemov je bilo že od nekdaj problematično. Danes je to še posebej evidentno pri pametnih mobilnih telefonih in tablicah. Z nedavnim razmahom teh naprav se je ponovno aktualiziralo vprašanje, kako učinkovito izrabiti omejeno strojno opremo. Za dobro izkoriščenost pomnilnika mora biti le-ta pravilno upravljan in recikliran, ko ni več v uporabi.

Nekateri, še danes aktualni, programski jeziki zahtevajo od programerja eksplicitno upravljanje s pomnilnikom. To neizbežno vpelje možnost programerskih napak, ki imajo lahko za posledico počasnejše delovanje programa (in morebiti celo sistema), pogosto pa tudi nepredvidljivo delovanje, kar privede do predčasnega končanja izvajanja programa ali napačnih rezultatov.

Z implementacijo čistilca pomnilnika za programski jezik C smo v večji meri odpravili napake, ki se pojavljajo pri eksplicitnem upravljanju pomnilnika in hkrati tudi zvišali stopnjo abstrakcije programskega jezika.

# Poglavje 2

# Motivacija

Abstrakcija je ključna razvojna smernica v računalništvu, ki je omogočila lažji in hitrejši razvoj programske opreme. Ko je bilo računalništvo še v povojih, se je programiralo ročno, bit po bitu, s pomočjo stikal. Kasneje so se pojavili programski jeziki, ki so močno poenostavili pisanje programov. Danes višji programski jeziki v okviru abstrakcije nudijo programerjem različne podatkovne konstrukte oziroma objekte, za katere prevajalniki samodejno rezervirajo prostor v pomnilniku. To opravljajo na tri načine [14]:

**Statično**

Pri tej metodi je objektom programa v času prevajanja ali povezovanja dodeljena pomnilniška lokacija, ki se med izvajanjem programa ne spreminja (je statična). Taka implementacija pa kljub dobri hitrosti izvajanja predstavlja nekaj omejitev za programski jezik:

1. Velikost objektov mora biti znana že v času prevajanja, torej ni mogoče ustvarjati dinamičnih objektov.
2. Funkcije ne omogočajo rekurzivnih klicev, saj so pomnilniške lokacije lokalnih objektov fiksne.

**Na skladu**

Sklad je del pomnilniškega prostora organiziranega po principu LIFO. Vsak klic funkcije potisne na vrh sklada svoj aktivacijski zapis, ki med

drugim vsebuje tudi prostor za lokalne spremenljivke. Ko se klic funkcije zaključi, se aktivacijski zapis odstrani z vrha sklada. Tak način delovanja sicer omogoča rekurzivne klice funkcij, vendar pa je življenjska doba objektov omejena na trajanje izvajanja funkcije - objekta ni možno vrniti starševski funkciji.

**V kopici**

Kopica, za razliko od sklada, nima nobene organizacijske strukture - objektom se lahko dodeli katerikoli del prostega pomnilnika znotraj kopice. Podpira ustvarjanje objektov dinamičnih velikosti, ki niso omejeni z življenjsko dobo funkcij. Med drugim s tem omogoča tudi funkcijske programske jezike, ki vračajo kot rezultat funkcije funkcijo. Toda vsa ta fleksibilnost kopice jo hkrati naredi tudi kompleksnejšo za upravljanje.

Danes večina višjih programskih jezikov podpira kopico. Ustvarjanje objektov je neproblematično - dvoumnosti ni, saj programer natanko na enem mestu eksplicitno zahteva rezervacijo prostora v kopici za objekt, ko ga potrebuje. Ker pa je pomnilnik, tako kot drugi viri računalnika, končen, ga je potrebno občasno počistiti odsluženih objektov. To lahko opravi eksplicitno (ročno) programer ali pa avtomatsko izvajalni sistem (*runtime system*).

## 2.1 Ekspliciten način čiščenja pomnilnika

Ekspliciten način čiščenja kopice nalaga breme sproščanja pomnilnika programerju, kar pa ga naredi dovzetnega za programske napake. Pojavljata se dva tipa napak:

**Viseči kazalci**

Ko se počisti del pomnilnika, ki ga je zasedal objekt, postanejo morebitni kazalci usmerjeni nanj viseči. Rezultat dostopa do pomnilnika preko visečega kazalca je nepredvidljiv. Lahko privede do sesutja programa, nepravilnih rezultatov, še najbolj verjetno pa se ne zgodi nič

neobičajnega; simptomi se morebiti pojavijo šele po mnogih procesorskih ciklih, vse to pa močno oteži razhroščevanje programa.

**Uhajanje pomnilnika**

Če program izgubi vse reference do objekta, ki še vedno zaseda pomnilnik, pride do uhajanja pomnilnika. Posledice so pogosto neopazne, lahko pa v skrajnem primeru že samo ena napaka sčasoma privede do izčrpanja pomnilniškega vira.

Očitno je, da ima nepravilno upravljanje s pomnilnikom lahko hude posledice za delovanje računalniškega sistema. Težavnost pravilne rabe eksplicitnega čiščenja pomnilnika izvira iz dejstva, da je živost objekta globalna lastnost [24], programski moduli pa morajo upravljati s pomnilnikom objekta v lokalnem kontekstu. V izogib napakam se zato v praksi tipično uporabljajo ustaljena pravila, ki se jih morajo programerji držati za pravilno upravljanje pomnilnika [5]. Pojavili so se tudi predlogi, ki rešujejo ta problem s pomočjo strojne opreme [18].

## 2.2 Avtomatski način čiščenja pomnilnika

Avtomatski način čiščenja pomnilnika razbremeni programerja eksplicitnega upravljanja s pomnilnikom. Ker čistilec pomnilnika (*garbage collector*) smatra za smeti le objekte do katerih program nima referenc, se problem visečih kazalcev ne more pojaviti. V večji meri odpravi tudi uhajanje pomnilnika, vendar le za objekte, ki so smeti. Če je objekt dosegljiv, toda ga program ne uporablja več, pomnilnik še vedno uhaja.

Uporaba čistilca pomnilnika pripomore tudi k večji abstrakciji programskih jezikov, saj ni več potrebe po eksplicitnih ukazih za delo s pomnilnikom. S tem se zmanjša soodvisnost med programskimi moduli, ker jim ni potrebno koordinirati upravljanje pomnilnika deljenih objektov z drugimi moduli, hkrati pa se poveča tudi zmožnost ponovne uporabe izvorne kode.

Glede na način delovanja delimo metode čiščenja pomnilnika v štiri osnovne skupine:

- označi-počisti (*mark-sweep*)
- označi-strni (*mark-compact*)
- prepisovanje (*copying*)
- štetje referenc (*reference counting*)

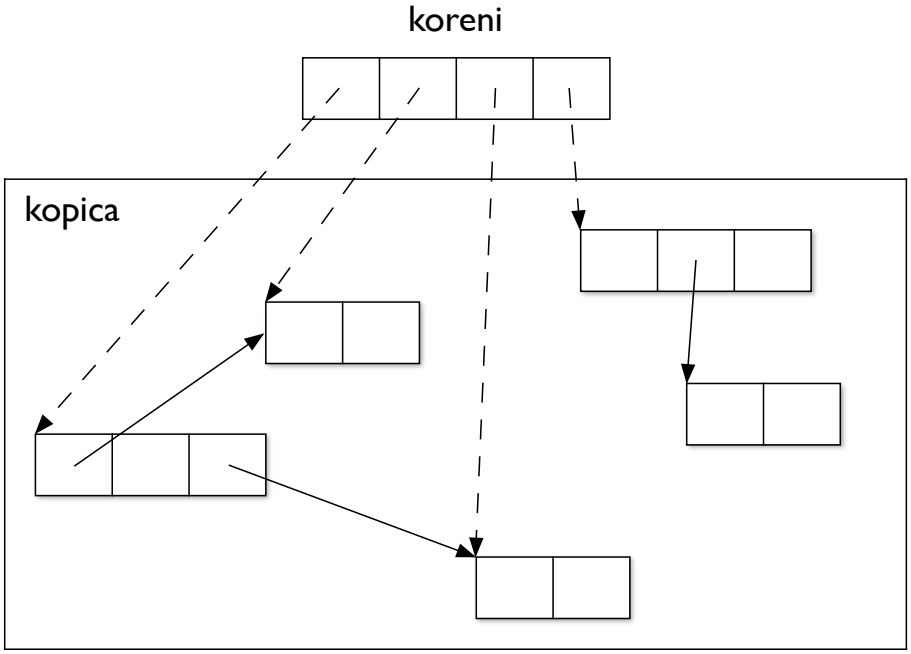
V praksi se danes večinoma uporabljajo čistilci pomnilnikov, ki kombinirajo več metod hkrati.

## 2.3 Definicije

Definicije so povzete po knjigi The Garbage Collection Handbook: The Art of Automatic Memory Management [13].

Objekt je definiran kot del pomnilnika v kopici, ki je bil dodeljen na zahtevo aplikacije. Sestavljen je iz polj, ta pa hranijo reference na druge objekte v kopici oziroma vrednosti primitivnih tipov (npr. celoštevilske).

Kopico se v kontekstu čiščenja pomnilnika interpretira kot usmerjen graf objektov (slika 2.1). Vozlišča predstavljajo objekte v kopici, povezave pa so reference na objekte, ki jih vsebuje (izvorni) objekt oziroma koren. Program dostopa do objektov v kopici preko izhodiščnih točk, ki jih imenujemo koreni; sem spadajo programu neposredno dostopne lokalne in globalne kazalčne spremenljivke ter registri.




Slika 2.1: Kopica kot usmerjen graf objektov.

Program s čistilcem pomnilnika se deli na dva dela [8]:

- mutator (*mutator*)
- čistilec (*collector*)

Mutator je del programa, ki izvaja kodo aplikacije. Aplikacija za doseg rezultatov ustvarja nove objekte ter spreminja reference v poljih objektov in korenih, s tem pa mutira graf objektov.

Čistilec je (impliciten) del programa, ki počisti pomnilnik nedosegljivih objektov - smeti. Ti so posledica izvajanja mutatorja.

Abstraktno izvajanje čistilca lahko razdelimo na dve ključni fazi [24]:

1. Določanje živih ali mrtvih objektov.
2. Reciklaža pomnilnika; sproščanje pomnilnika mrtvih objektov za ponovno uporabo.

Ti dve fazi nista neodvisni in se pogosto tudi prepletata.

Čistilec deluje pravilno le, če ne počisti živih objektov. V osnovni definiciji je objekt v nekem trenutku živ, če se bo v prihodnosti do njega še dostopalo. Ker pa je s tako definicijo nemogoče določiti živost objekta med izvajanjem, se v praksi uporablja ohlapnejšo definicijo; objekt se v nekem trenutku smatra kot živ, če do njega vodi vsaj ena pot preko kazalcev z začetkom v korenu - torej če je objekt (posredno ali neposredno) dostopen mutatorju. Takšna definicija zagotavlja pravilno delovanje čistilca, saj so vsi objekti, do katerih ni moč dostopati, zagotovo mrtvi. Po drugi strani pa uvršča med žive objekte tudi tiste, do katerih program sicer ne bo več dostopal (po osnovni definiciji mrtve). Raziskave so pokazale, da se lahko z natančnejšo opredelitvijo živih objektov, ob pomoči prevajalnika, občutno izboljša učinkovitost čiščenja [12].

# Poglavje 3

# Algoritmi za čiščenje pomnilnika

Vsi algoritmi (opis in psevdokoda), ki so predstavljeni v tem poglavju, so povzeti po The Garbage Collection Handbook: The Art of Automatic Memory Management [13].

Vsi osnovni algoritmi delujejo po principu „ustavi svet“ (*stop-the-world*). To pomeni, da so med izvajanjem čistilca vse niti mutatorja ustavljene - s stališča mutatorja je čiščenje izvedeno atomarno. Na ta način se za čas izvajanja čistilca graf objektov zamrzne.

## 3.1 Označi-počisti

Algoritem označi-počisti je bil prvi algoritem razvit za avtomatsko čiščenje pomnilnika [16]. Implementiran je bil kot del programskega jezika Lisp. Ideja algoritma neposredno sledi abstraktnim fazam izvajanja čistilca:

1. Označevanje (*marking*): določanje živih objektov. Algoritem se sprehodi čez graf objektov z začetkom v korenih in označi vsak obiskani objekt.

2. Pometanje (*sweeping*): reciklaža mrtvih objektov. Vsak objekt, ki ni bil označen iz prejšnje faze (torej ni dostopen mutatorju), je mrtev in njegov pomnilniški prostor se sprosti.

### 3.1.1 Algoritem označi-počisti

Komunikacija med mutatorjem in čistilcem je minimalna; v primeru, da mutator neuspešno ustvari nov objekt, le-ta sproži čistilca preko klica procedure `COLLECT`. Če je ustvarjanje novega objekta ponovno neuspešno, je kopica izčrpana.

**Algorithm 1** Dodeljevanje pomnilnika.

```
 1: procedure NEW()
 2:     ref ← ALLOCATE()
 3:     if ref = null then
 4:         COLLECT()
 5:         ref ← ALLOCATE()
 6:         if ref = null then                    ▷ Kopica je izčrpana
 7:             error: Memory exhausted
 8:     return ref

 9: procedure COLLECT()
10:     MARKFROMROOTS()
11:     SWEEP(HeapStart, HeapEnd)
```

Faza označevanja se začne s pripravo delovnega seznama (*worklist*). Vse objekte, ki jih neposredno referencirajo koreni mutatorja, se označi in doda v delovni seznam. Da se zmanjša velikost le-tega, se takoj po dodajanju vsakega objekta kliče procedura `MARK`.

`MARK` jemlje objekte iz delovnega seznama (ti so že označeni) ter pregleduje njihova kazalčna polja. Če polje kaže na neoznačen objekt, se ga označi in doda v delovni seznam. Izvajanje rutine se ustavi, ko se seznam izprazni.

Če se delovni seznam implementira kot sklad, poteka preiskovanje v globino. V primeru, da so označevalni biti shranjeni v objektih, se s tem izboljša lokalnost pomnilniških dostopov; naslednji objekt iz seznama je bil najverjetneje pred kratkim označen in še vedno v predpomnilniku.

**Algorithm 2** Označevanje.

```
 1: procedure markFromRoots()
 2:     initialise(worklist)
 3:     for each fld in Roots do
 4:         ref ← *fld
 5:         if ref ≠ null and not isMarked(ref) then
 6:             setMarked(ref)
 7:             add(worklist, ref)
 8:             mark()

 9: procedure initialise(worklist)
10:     worklist ← empty

11: procedure mark()
12:     while not isEmpty(worklist) do
13:         ref ← remove(worklist)
14:         for each fld in Pointers(ref) do
15:             child ← *fld
16:             if child ≠ null and not isMarked(child) then
17:                 setMarked(child)
18:                 add(worklist, child)
```

Sledi še zaključna faza; procedura `SWEEP` linearno počisti pomnilnik vseh mrtvih objektov in živim objektom obrne označevalni bit za pripravo na naslednje označevanje.

**Algorithm 3** Pometanje.

```
1: procedure SWEEP(start, end)
2:     scan ← start
3:     while scan < end do
4:         if ISMARKED(scan) then
5:             UNSETMARKED(scan)
6:         else
7:             FREE(scan)
8:         scan ← NEXTOBJECT(scan)
```

### 3.1.2 Izboljšave

Ena izmed izboljšav je označevanje s t.i. bitnim poljem (*bitmap table*). Označevalne bite iz objekta se premakne v ločeno tabelo, kjer vsak bit povezuje točno določen naslov v kopici, kamor je možno shraniti objekt. Ker ni potrebe po pisanju v objekt za spreminjanje označevalnega bita, se s tem umaže manj predpomnilniških vrstic in je zato posledično tudi manj pisanja v pomnilnik. V praksi se izkaže, da objekti tipično živijo in umirajo v gručah [10, 15], kar omogoča v fazi pometanja preverjanje več bitov hkrati v tabeli (saj so vrednosti bitov sosednjih objektov z veliko verjetnostjo enaki) in na enak način tudi hitrejšo reciklažo mrtvih objektov (sproščanje pomnilnika v gručah).

## 3.2 Označi-strni

Algoritem označi-počisti ne preureja objektov v kopici, kar sčasoma privede do razdrobljenosti prostega prostora med žive objekte - zunanja fragmentacija. Težave se pojavijo pri dodeljevanju pomnilnika za nove objekte, ker je potrebno najti dovolj velik nepretrgan prazen prostor. Če je stopnja fragmentiranosti visoka, lahko pri ustvarjanju velikega objekta pride do izčrpanosti kopice kljub temu, da bi bila akumulirana vsota praznih koščkov pomnilnika dovolj velika za objekt.

Za minimizacijo fragmentacije se uporabljajo rešitve, ki združujejo objekte podobnih velikosti skupaj v bloke. Težavo pa povsem odpravijo čistilci, ki strnjujejo žive objekte.

Prva taka skupina algoritmov se imenuje označi-strni. Žive objekte strnejo na en konec kopice, kar naredi dodeljevanje pomnilnika hitro, analogno skladu; potrebno je le preveriti, če je dosežen konec kopice in povečati kazalec za velikost dodelitve. Ker ni potrebno iskati prostega prostora, se zmanjša čas procesiranja in brez fragmentacije se zmanjša tudi potratnost pomnilniškega prostora.

Algoritmi označi-strni, podobno kot označi-počisti, začnejo s fazo označevanja, v kasnejših fazah pa nato strnjujejo objekte s premikanjem le-teh po kopici in posodabljanjem kazalcev, da kažejo na nove lokacije objektov. Algoritmi se pri razvrščanju objektov ravnajo po treh strategijah:

**Arbitrarno**

Objekte se razporeja brez pravil. Izvajanje je hitro, vendar je zaradi naključne razporeditve objektov prostorska lokalnost slaba.

**Linearno**

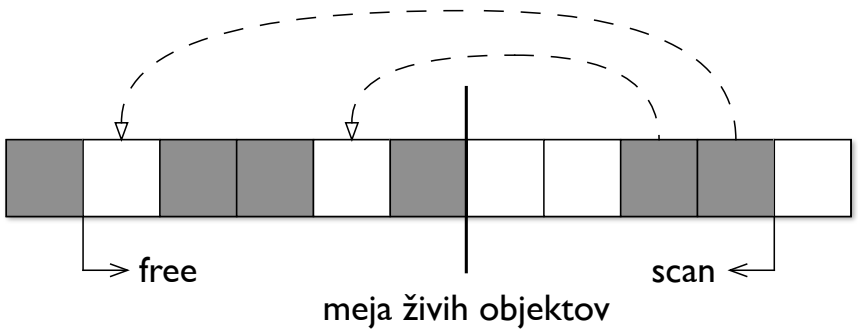
Objekte se razporeja po skupnih lastnostih (npr. če so si blizu v grafu objektov - starši in otroci) za izboljšanje lokalnosti. Tipično se uporablja pri prepisovalnih algoritmih.

**Drsno**

Objekte se podrsa skupaj na en konec kopice. Ohrani se prvotni vrstni red objektov v pomnilniku in s tem tudi lokalnost.

### 3.2.1 Algoritem dveh prstov

Edwardov algoritem dveh prstov (*Two Finger algorithm*) [22] strnjuje objekte enake velikosti v regiji kopice, določene z dvema kazalcema: `free` kaže na začetek regije in `scan` kaže na konec regije. Zaradi enakih velikosti objektov se lahko vnaprej izračuna mejo, do kje bodo živi objekti segali v regiji po strnjevanju. Izvajanje nato poteka v dveh prehodih:




Slika 3.1: Edwardov algoritem dveh prstov. Sivi prostori predstavljajo objekte, beli pa prazen proster. Objekte nad mejo (`scan`) se prepiše na mesta pod mejo (`free`) (slika je povzeta iz [13]).

1. V prvem prehodu se premika objekte nad mejo na prazna mesta pod mejo. Procedura `RELOCATE` najprej povečuje kazalec `free` do prvega prostega mesta oziroma mrtvega objekta in nato zmanjšuje kazalec `scan` do prvega živega objekta (slika 3.1). Če se kazalca preideta, je prehoda konec. Sicer pa se prepiše objekt, kamor kaže `scan`, na mesto kamor kaže `free`. Po tem se na mestu `scan` zapiše nov naslov objekta, ki se uporabi ob naslednjem prehodu.

2. V drugem prehodu se posodobi vse kazalce, ki kažejo na objekte nad mejo (ti so bili premaknjeni) z novimi lokacijami zapisanimi v starih lokacijah objektov.

### 3.2.2 Izboljšave

Algoritem razporeja objekte arbitrarno, kar implicira slabo prostorsko lokalnost. Ker pa objekti pogosto živijo in umirajo v gručah, se jih lahko v gručah tudi premika na prazne prostore pod mejo in posledično izboljša lokalnost mutatorja.

**Algorithm 4** Two-Finger.

```
 1: procedure COMPACT()
 2:     RELOCATE(HeapStart, HeapEnd)
 3:     UPDATEREFERENCES(HeapStart, free)

 4: procedure RELOCATE(start, end)
 5:     free ← start
 6:     scan ← end
 7:     while free < scan do
 8:         while ISMARKED(free) do
 9:             UNSETMARKED(free)
10:             free ← free + SIZE(free)
11:         while not ISMARKED(scan) and scan > free do
12:             scan ← scan − SIZE(scan)
13:         if scan > free then
14:             UNSETMARKED(scan)
15:             MOVE(scan, free)
16:             *scan ← free
17:             free ← free + SIZE(free)
18:             scan ← scan − SIZE(scan)

19: procedure UPDATEREFERENCES(start, end)
20:     for each fld in Roots do
21:         ref ← *fld
22:         if ref ≥ end then
23:             *fld ← *ref
24:
25:     scan ← start
26:     while scan < end do
27:         for each fld in POINTERS(scan) do
28:             ref ← *fld
29:             if ref ≥ end then
30:                 *fld ← *ref
31:         scan ← scan + SIZE(scan)
```

## 3.3 Prepisovanje

Druga skupina algoritmov za strnjevanje namesto preurejanja objektov znotraj kopice, le-te prepisuje v ločen del. Ti algoritmi ohranjajo izboljšave čistilcev označi-strni, hkrati pa ne zahtevajo večkratnega prehoda čez kopico. S tem se izboljša hitrost čiščenja, a s hudo prostorsko ceno - velikost kopice se prepolovi.

Bistvo čistilca je prepisovanje živih objektov iz enega dela kopice v drug del. Za razliko od algoritmov označi-strni, ki ločijo fazo označevanja, se pri prepisovalnih algoritmih prepoznavanje živih objektov in čiščenje (t.j. prepisovanje) izvedeta hkrati. Ker na velik delež objektov v programu kaže le en kazalec [21], se v večini primerov objekte obišče le enkrat.

### 3.3.1 Polprostorsko prepisovanje

Algoritem polprostorskega prepisovanja (*semispace copying*) [9] razpolovi kopico na dva (enako velika) polprostora; izvorni prostor (*fromspace*) in ponorni prostor (*tospace*). Dodeljevanje pomnilnika vedno poteka v ponornem prostoru, na enak način kot pri označi-strni čistilcih oziroma skladu - potrebno je le povečevanje števca za velikost dodelitve. Ko je ponorni prostor izčrpan, se sproži čiščenje pomnilnika.

Procedura `COLLECT` najprej zamenja vlogi polprostorov - ponorni prostor postane izvorni in obratno. Nato se izvede prepisovanje živih objektov iz izvornega prostora v ponornega. Vse žive objekte se strni na začetek ponornega prostora, izvorni prostor pa zavrže do naslednjega čiščenja. Dodeljevanje pomnilnika nato poteka normalno v ponornem prostoru (od zadnjega prepisanega objekta naprej) do naslednjega čiščenja.

Prepisovanje se prične z evakuacijo korenskih objektov v ponorni prostor, kar hkrati te objekte tudi implicitno doda v delovni seznam. Objekte se nato jemlje iz seznama in njihova kazalčna polja procesira na sledeč način; če je objekt, kamor kaže kazalec polja, v izvornem prostoru, se ga prepiše v ponorni prostor (in s tem implicitno doda v delovni seznam) ter vstavi naslov kopije

**Algorithm 5** Inicializacija kopice in dodeljevanje pomnilnika.

```
1:  procedure createSemispaces()
2:      tospace ← HeapStart
3:      extent ← (HeapEnd − HeapStart)/2
4:      top ← fromspace ← HeapStart + extent
5:      free ← tospace

6:  procedure allocate(size)
7:      result ← free
8:      newfree ← result + size
9:      if newfree > top then
10:         return null                    ▷ Ponorni prostor je izčrpan
11:     free ← newfree
12:     return result
```

v originalen objekt (iz izvornega prostora) kot posredovalni naslov. Če pa je objekt že bil prepisan, se prebere posredovalni naslov iz objekta v izvornem prostoru, kamor kaže kazalec polja. Naslov kopije iz ponornega prostora se nato vpiše v kazalčno polje namesto starega naslova. S tem procesom se žive objekte prepiše le enkrat in hkrati posodobi vse reference nanje.

Delovni seznam je po Cheneyevem algoritmu [6] implementiran kot vrsta FIFO, kar implicira preiskovanje grafa objektov v širino. Za realizacijo sta potrebna dva kazalca; `free` in `scan`. Oba se pred začetkom prepisovanja nastavi na začetek ponornega prostora. `free` kaže tik za zadnjim živim objektom v ponornem prostoru na mesto, kamor se bo prepisal naslednji objekt, `scan` pa kaže na naslednji objekt delovnega seznama za procesiranje v ponornem prostoru. Ko `scan` doseže kazalec `free`, je delovni seznam prazen in graf objektov je preiskan - vsi živi objekti so evakuirani v ponorni prostor in izvajanje čistilca je končano.

Slika 3.2 prikazuje na primeru delovanje Cheneyevega algoritma. Najprej se prepiše korenske objekte (mutatorju neposredno dostopne) v ponorni pro-

**Algorithm 6** Polprostorsko prepisovanje.

```
1:  procedure COLLECT()
2:      FLIP()
3:      INITIALISE(worklist)
4:      for each fld in Roots do
5:          PROCESS(fld)
6:      while not isEMPTY(worklist) do
7:          ref ← REMOVE(worklist)
8:          SCAN(ref)

9:  procedure FLIP()
10:     fromspace, tospace ← tospace, fromspace        ▷ Zamenja vrednosti
11:     top ← tospace + extent
12:     free ← tospace

13: procedure SCAN(ref)
14:     for each fld in POINTERS(ref) do
15:         PROCESS(fld)

16: procedure PROCESS(fld)
17:     fromRef ← *fld
18:     if fromRef ≠ null then
19:         *fld ← FORWARD(fromRef)      ▷ Zapiše naslov tospace objekta

20: procedure FORWARD(fromRef)
21:     toRef ← FORWARDINGADDRESS(fromRef)
22:     if toRef = null then                          ▷ Objekt še ni prepisan
23:         toRef ← COPY(fromRef)
24:     return toRef
```

**Algorithm 7** Polprostorsko prepisovanje (nad.).

```
1: procedure COPY(fromRef)
2:     toRef ← free
3:     free ← free + SIZE(fromRef)
4:     MOVE(fromRef, toRef)
5:     FORWARDINGADDRESS(fromRef) ← toRef
6:     ADD(worklist, toRef)
7:     return toRef
```

**Algorithm 8** Cheneyev delovni seznam.

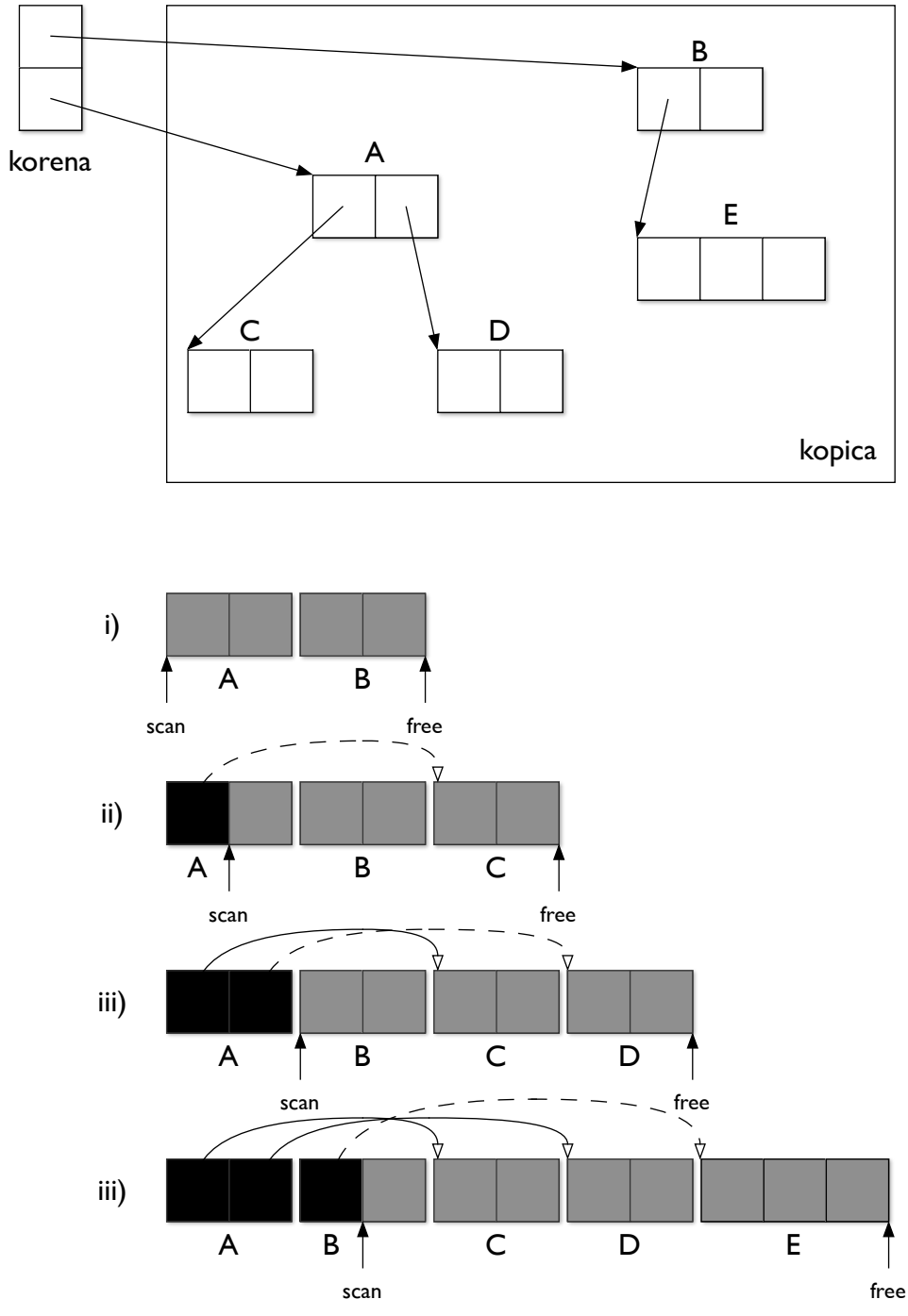
```
1: procedure INITIALISE(worklist)
2:     scan ← free

3: procedure ISEMPTY(worklist)
4:     return scan = free

5: procedure REMOVE(worklist)
6:     ref ← scan
7:     scan ← scan + SIZE(scan)
8:     return ref

9: procedure ADD(worklist, ref)                ▷ Ni potrebno ničesar
10:    /* nop */
```

stor (i). Še ne procesirana polja objektov se označuje s sivo barvo. Kazalec `scan` kaže na prvo kazalčno polje objekta A, ki nadaljnjo kaže na objekt C v izvornem prostoru. Ker objekt C ne vsebuje posredovalnega naslova, se ga prepiše na mesto, kamor kaže kazalec `free`. Kazalčno polje objekta A se posodobi na kopijo objekta C v ponornem prostoru in pobarva črno. Kazalec `scan` se premakne na naslednje polje objekta, `free` pa na prvo prosto mesto po prepisanem objektu (ii). Izvajanje se nadaljuje po enakem vzorcu, dokler se kazalca ne srečata oziroma so vsa polja obarvana črno.




Slika 3.2: Potek Cheneyevega algoritma na primeru (slika je povzeta iz [24]).

### 3.3.2 Izboljšave

Organizacija objektov znotraj kopice ima lahko nezanemarljiv vpliv na hitrost izvajanja mutatorja. Arbitrarna strategija razvrščanja objektov občutno poslabša prostorsko lokalnost pomnilniških dostopov mutatorja [4]. Po drugi strani pa je drsenje objektov konzervativne narave, ki ne spreminja vzorcev lokalnosti mutatorja. Toda s prepisovanjem objektov v ločen prazen prostor se lahko uvede tudi kompleksnejše urejanje objektov z namenom izboljšanja lokalnosti.

V praksi idealnega urejanja ni mogoče udejanjiti. Četudi bi poznali dostope do pomnilnika v prihodnosti, je še vedno problem popolne ureditve NP-poln [20]. Iz tega razloga se uporablja različne hevristike za vodenje razporejanja. Ena izmed teh poskuša postaviti otroke ob enem izmed staršev, saj se dostop do objektov vrši preko staršev. Tako hevristiko se lahko doseže s preiskovanjem grafa objektov v globino.

Manj prostorsko zahtevna oblika od pravega preiskovanja v globino s Fenichel in Yochelsonovim algoritmom predstavlja Moonova [17] modifikacija Cheneyevega algoritma, ki izvede približek preiskovanja v globino. Po prepisanju objekta se namesto kazalca `scan` uporabi `partialScan` za obdelavo zadnje strani objektov. Ko se kazalca `partialScan` in `free` dosežeta, se ponovno uporabi kazalec `scan`.

Ker pa si algoritem ne zapomni že obdelanih objektov s kazalcem `partialScan`, se okoli 30% objektov ponovno obišče z glavno, linearno obdelavo delovnega seznama (s kazalcem `scan`) [25]. Rešitev predstavlja algoritem, ki za vsako stran v pomnilniku hrani posebej instanci kazalcev `scan` in `partialScan`.

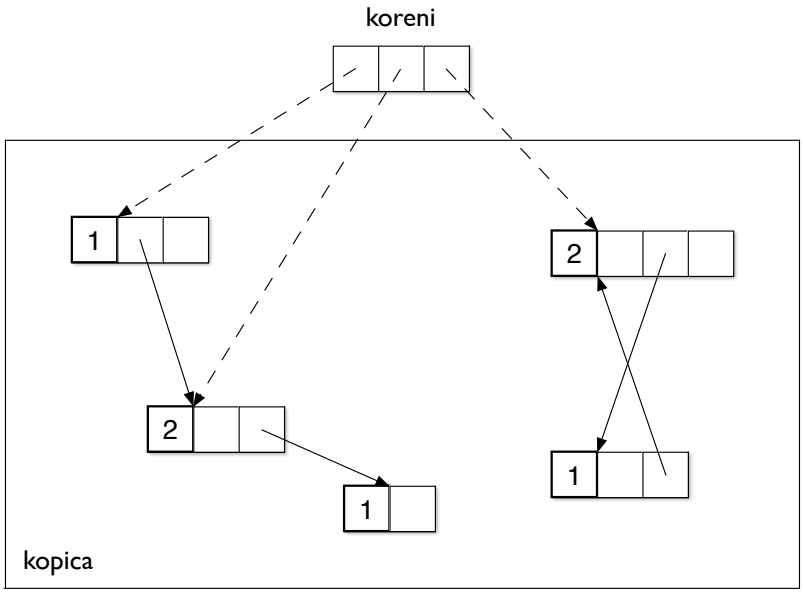
**Algorithm 9** Moonova modifikacija Cheneyevega algoritma.

```
1:  procedure INITIALISE(worklist)
2:      scan ← free
3:      partialScan ← free

4:  procedure ISEMPTY(worklist)
5:      return scan = free

6:  procedure REMOVE(worklist)
7:      if partialScan < free then
8:          ref ← partialScan
9:          partialScan ← partialScan + SIZE(partialScan)
10:     else
11:         ref ← scan
12:         scan ← scan + SIZE(scan)
13:     return ref

14: procedure ADD(worklist, ref)
15:     partialScan ← MAX(partialScan, STARTOFPAGE(ref))
```




Slika 3.3: Primer štetja referenc. Števec referenc v osnovni implementaciji beleži število vseh referenc na objekt.

## 3.4 Štetje referenc

Štetje referenc se pri načinu razpoznavanja mrtvih objektov temeljno razlikuje od ostalih osnovnih metod čiščenja pomnilnika. Namesto posrednega odkrivanja živih objektov s preiskovanjem grafa objektov, se živost objekta določi neposredno iz objekta samega.

Sistem štetja referenc za vsak objekt beleži število referenc nanj (slika 3.3). Število se poveča, če se ustvari nov kazalec na objekt in zmanjša, če se kazalec nanj izbriše. Objekt se smatra kot živ samo v primeru, da je število referenc večje od nič. Tipično se ta vrednost hrani v glavi objekta skupaj z drugimi metapodatki.

Štetje referenc se uporablja v programskih jezikih (Perl, PHP, Python . . . ), v specifičnih aplikacijah (npr. Adobe Photoshop) in pogosto tudi v da-

totečnih sistemih (npr. Unix). Rešitev prinaša številne prednosti:

- Delo čiščenja je prepleteno z mutatorjem, zato se stroški režije upravljanja pomnilnika enakomernejše porazdelijo tekom izvajanja programa. To je pomembna lastnost za npr. interaktivna okolja in realno-časovne sisteme, kjer je dopustno le kratko ustavljanje programa.

- V nasprotju z drugimi metodami najpogosteje nima negativnih vplivov na lokalnost pomnilniških dostopov - števce referenc se spreminja objektom, do katerih mutator dostopa.

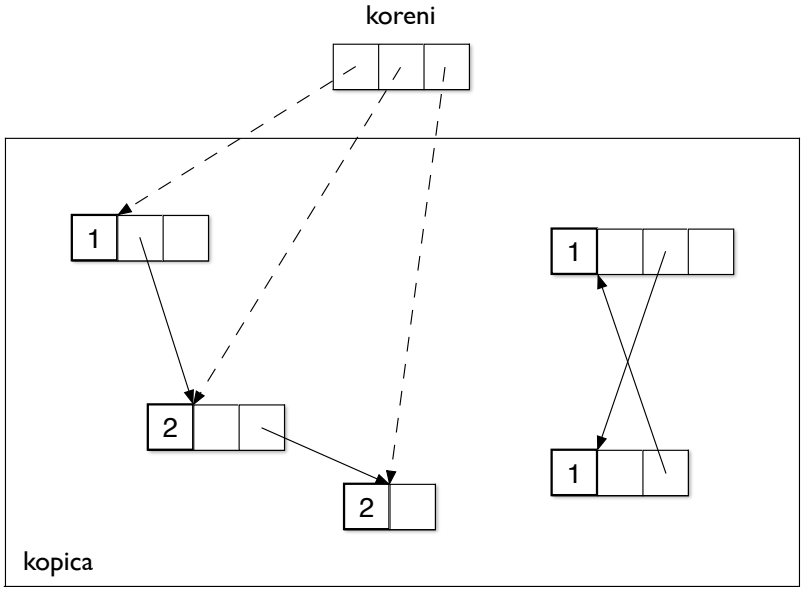
- Takojšnje razpoznavanje mrtvih objektov (in reciklaža) omogoča hitro delovanje sistema tudi v okoljih z visoko zasedenostjo kopice. Druge rešitve bi bile primorane v takih okoliščinah pogosto izvajati (drago) čiščenje.

Ima pa štetje referenc tudi nekaj slabih lastnosti:

- Stroški režije čiščenja so višji v primerjavi z ostalimi metodami. To velja še posebej za naivno implementacijo, ki sledi vsem spremembam referenc; vsaka posodobitev kazalčne spremenljivke iz stare na novo referenco zahteva dostop do obeh objektov za posodobitev števca referenc (povečati novo vrednost in zmanjšati staro).

- Osnovna implementacija ne predvideva reciklaže cikličnih objektnih struktur. To so množice objektov, ki preko referenc tvorijo usmerjen cikel v grafu objektov. Števci referenc objektov so v taki strukturi večji od nič, četudi niso dosegljivi mutatorju (slika 3.4) - prihaja do uhajanja pomnilnika.

- Shranjevanje števca referenc poveča pomnilniško zasedbo objekta. V (teoretično) najslabšem primeru je lahko vrednost števca referenc enaka številu objektov v kopici, zato mora biti velikost števca enaka velikosti kazalca. Ker pa je to v praksi zelo redek pojav, zadošča že majhen števec z dodano logiko, ki upravlja s prekoračitvami števca.




Slika 3.4: Primer ciklične objektne strukture pri štetju referenc. V kopici na desni strani sta objekta živa, čeprav mutatorju nista dostopna.

- V določenih primerih lahko pride do daljših pavz pri reciklaži objektov; v primeru, da umre objekt z veliko kazalčnimi polji, je potrebno primerno posodobiti in/ali reciklirati vse potomce.

### 3.4.1 Preprost sistem štetja referenc

Naivna implementacija dosledno sledi vsem spremembam referenc na objekte. Ob posodobitvi vrednosti pomnilniške lokacije procedura `WRITE` poveča števec referenc objektu, katerega referenco se vpisuje in zmanjša objektu, ki izgubi referenco. Implementacija predvideva posodabljanje tudi za lokalne spremenljivke, zato se jim mora pred vrnitvijo funkcije vpisati vrednost `null` za pravilno posodobitev števca referenc.

**Algorithm 10** Preprost sistem štetja referenc.

```
 1: procedure New()
 2:     ref ← allocate()
 3:     if ref = null then                          ▷ Kopica je izčrpana
 4:         error: Memory exhausted
 5:     rc(ref) ← 0
 6:     return ref

 7: procedure Write(src, i, ref)
 8:     addReference(ref)                           ▷ Vrstni red je pomemben
 9:     deleteReference(src[i])
10:     src[i] ← ref

11: procedure addReference(ref)
12:     if ref ≠ null then
13:         rc(ref) ← rc(ref) + 1

14: procedure deleteReference(ref)
15:     if ref ≠ null then
16:         rc(ref) ← rc(ref) − 1
17:         if rc(ref) = 0 then
18:             for each fld in Pointers(ref) do
19:                 deleteReference(∗fld)
20:             free(ref)
```

### 3.4.2 Izboljšave

Prva v vrsti izboljšav osnovnega algoritma je t.i. zakasnjeno štetje referenc (*deferred reference counting*) [7]. Programi v večini primerov nalagajo kazalce v lokalne ali začasne spremenljivke (registre), zato se lahko z odstranitvijo manipulacij števcev referenc v teh primerih občutno zmanjša stroške režije čiščenja. Zakasnjeno štetje referenc upošteva le nalaganje referenc v polja objektov, ostale pa ignorira. Števec referenc zato ne odraža več dejanskega števila referenc, temveč le število referenc iz objektov v kopici. Ko pade število referenc objekta na nič, posledično ni več možna takojšnja reciklaža. Potrebno je periodično posodabljanje števcev referenc na prave vrednosti (torej upoštevajoč tudi lokalne in začasne spremenljivke), da se lahko reciklaža izvede pravilno.

Procedura `WRITE` izvrši nalaganje kazalcev v korene mutatorja brez manipulacij števcev referenc. Za ostala nalaganja (v polja objektov) pa se vrši podobno izvajanje kot v osnovnem algoritmu. `DELETEREFERENCETOZCT` se kliče za zmanjševanje števca referenc; če ta pade na nič, doda objekt v `ZCT`. `ZCT` (tabela) tako vsebuje kandidate za reciklažo. Za sproščanje pomnilnika se periodično proži procedura `COLLECT`, ki neposredno dostopnim objektom iz korenov posodobi števce referenc na prave vrednosti. Vse objekte v `ZCT`, ki imajo števec referenc še vedno enak nič, se lahko nato varno reciklira.

Problem referenčnih ciklov se v praksi rešuje na več načinov. Ob štetju referenc se lahko uporabi tudi sekundarni čistilec, ki deluje na osnovi označevanja živih objektov za reciklažo cikličnih struktur. Nerodnejša rešitev obstaja tudi v prilagojenem stilu programiranja, ki se izogiba ustvarjanju takšnih struktur. Pojavile so se tudi druge rešitve [19], vendar pa se zaradi slabe hitrosti v praksi niso prijele.

**Algorithm 11** Zakasnjeno štetje referenc.

```
1:  procedure New()
2:      ref ← allocate()
3:      if ref = null then
4:          collect()
5:          ref ← allocate()
6:          if ref = null then                          ▷ Kopica je izčrpana
7:              error: Memory exhausted
8:      rc(ref) ← 0
9:      add(zct, ref)
10:     return ref

11: procedure Write(src, i, ref)
12:     if src = Roots then
13:         src[i] ← ref
14:     else
15:         addReference(ref)                           ▷ Vrstni red je pomemben
16:         remove(zct, ref)
17:         deleteReferenceToZCT(src[i])
18:         src[i] ← ref

19: procedure deleteReferenceToZCT(ref)
20:     if ref ≠ null then
21:         rc(ref) ← rc(ref) − 1
22:         if rc(ref) = 0 then
23:             add(zct, ref)

24: procedure collect()
25:     for each fld in Roots do
26:         addreference(∗fld)
27:     sweepZCT()
28:     for each fld in Roots do
29:         deleteReferenceToZCT(∗fld)
```

**Algorithm 12** Zakasnjeno štetje referenc (nad.).

```
1: procedure SWEEPZCT()
2:     while not ISEMPTY(zct) do
3:         ref ← REMOVE(zct)
4:         if RC(ref) = 0 then
5:             for each fld in POINTERS(ref) do
6:                 DELETEREFERENCE(*fld)
7:             FREE(ref)
```

# Poglavje 4

# Implementacija konzervativnega čistilca za programski jezik C

V sledečem poglavju je predstavljena lastna implementacija čistilca pomnilnika za programski jezik C.

## 4.1 Specifikacija čistilca

Za implementacijo čistilca je potrebno določiti naslednje specifikacije:

- V katerem programskem jeziku bo čistilec uporabljen.
- Kateri programski jezik izbrati za implementacijo čistilca.
- Katero vrsto čistilca implementirati.

### 4.1.1 Programski jezik C

Standardi programskega jezika C ne vključujejo čistilca pomnilnika. Namesto tega se s pomnilnikom eksplicitno upravlja z uporabo razreda funkcij `malloc`

in `free`. To ga uvršča v enega izmed redkih, še danes aktualnih, jezikov brez avtomatskega čiščenja pomnilnika.

Programski jezik C dopušča veliko fleksibilnosti (npr. kazalčna aritmetika), kar ga naredi manj abstraktnega in primernejšega za nizkonivojsko upravljanje z računalniškimi viri. Zaradi minimalne stopnje balasta je njegova uporaba močno razširjena v sistemskem programiranju.

Zaradi naštetih lastnosti smo se odločili implementirati čistilca pomnilnika v programskem jeziku C, za uporabo v taistem jeziku. Uspešna vpeljava čistilca za programski jezik C (in C++) sicer že obstaja za vrsto platform pod imenom Boehm-Demer-Weiser [1], ki se razvija in dopolnjuje že več kot 20 let. Lastna implementacija predstavlja zato pretežno pedagoško vrednost.

### 4.1.2 Konzervativni čistilec pomnilnika

Programski jeziki zasnovani z avtomatskim čiščenjem pomnilnika tipično uporabljajo tipsko natančne sisteme (*type-accurate system*). Ti prepoznajo vse kazalce, kjer se lahko nahajajo. Nasprotno pa čistilci za nekooperativne jezike, kot je jezik C, ne prejmejo informacij o tipih podatkov od prevajalnika ali izvajalnega sistema. Za pravilno delovanje so primorani sprejemati konzervativne odločitve o vsebnosti kazalcev v korenih in poljih objektov - če je vsebovana vrednost dovolj podobna kazalcu, se jo smatra kot kazalec (četudi to morda ni). Od tod tudi ime konzervativni čistilec. Lahko se zgodi, da naključno zaporedje bitov tvori legitimen naslov v kopici in se zato objekt napačno smatra kot živ.

Konzervativno prepoznavanje kazalcev ima za delovanje čistilca dve pomembni implikaciji:

1. Čistilec ne sme spreminjati vrednosti korenov in polj objektov. Ta omejitev onemogoča delovanje vseh čistilcev, ki premikajo objekte v kopici, saj ne morejo posodobiti referenc nanje. Podobno se tudi označevalni biti ne morejo hraniti v glavah objektov, ker se lahko z napačno interpretacijo kazalca sledi objektu, ki to sploh ni, in s pisanjem v pomnilnik pokvari uporabnikove podatke.

2. Zaradi svobode pri delu s kazalci v programskem jeziku C je dobro minimizirati možnost dostopa mutatorja do čistilčevih metapodatkov. Te se zato premakne iz glav objektov v ločen prostor.

Izbira čistilca se je zaradi navedenih omejitev zožila na tip označi-počisti, ki konzervativno prepoznava kazalce in hrani označevalne bite ločeno od objektov v bitnem polju.

## 4.2 Uporaba čistilca

Zaradi specifik implementacije je delovanje čistilca omejeno na računalniško okolje Mac OS X (x86-64). Za uporabo je potrebno priskrbeti dve datoteki:

- Zaglavno datoteko (*header file*): `garbagecollector.h`
- Izvorno datoteko (*source file*): `garbagecollector.c`

Izvorna datoteka mora imeti prisotno tretjerazredno knjižnico `uthash` [3]. Zaglavna datoteka definira programski vmesnik čistilca, ki ga potrebuje prevajalnik za pravilno tvorbo kode. Vsebuje deklaracije vseh funkcij, ki so na voljo uporabnikom čistilca (slika 4.1):

- `gc_malloc(size_t size)`: Rezervira pomnilniški prostor velikosti `size`, ki ga bo upravljal čistilec. Vrnjenega kazalca se ne sme eksplicitno sprostiti s klicem funkcije `free()`.
- `gc_calloc(size_t num_items, size_t size)`: Rezervira dovolj velik pomnilniški prostor za `num` objektov velikosti `size`, ki ga bo upravljal čistilec. Prostor se napolni z bajti vrednosti 0. Vrnjenega kazalca se ne sme eksplicitno sprostiti s klicem funkcije `free()`.
- `gc_init()`: Inicializira čistilca pomnilnika. Klic se mora opraviti le enkrat, pred prvo uporabo funkcije `gc_malloc(size_t size)`.
- `gc_collect()`: Sproži čiščenje pomnilnika.

```
/*********  Alloctor    *********************/
/**
 * Allocates a new garbage-collected memory block of size size.
 * Pointer returned should not be freed explicitly.
 */
void *gc_malloc(size_t size);

/**
 * Allocates a new garbage-collected memory block of size size;
 * block is cleared.
 * Pointer returned should not be freed explicitly.
 */
void *gc_calloc(size_t num_items, size_t size);

/*********  Garbage collector   ************/
/**
 * Initializes garbage collector.
 * Must be called before allocating memory with gc_malloc().
 */
void gc_init();

/**
 * Initiates garbage collection.
 */
void gc_collect();
```

Slika 4.1: Zaglavna datoteka `garbagecollector.h`

Čiščenje pomnilnika se sicer samodejno izvede po določenem številu dodeljevanj pomnilnika, lahko pa ga tudi ročno prožimo s klicem funkcije `gc_collect()`.

Implementacija omogoča, da se lahko ob avtomatskem upravljanju pomnilnika uporablja tudi eksplicitno upravljanje. Pomembno je le, da ne kličemo funkcije `free()` nad pomnilniškim blokom, ki je upravljan s strani čistilca.

Zaradi podrobnosti implementacije čistilca smo morali vpeljati naslednje omejitve programskega jezika C:

- Veljaven kazalec na blok pomnilnika je tisti, ki ga kot rezultat funkcije vrne `gc_malloc(size_t size)` oziroma `gc_calloc(size_t num_items, size_t size)`. Če kazalec kaže na drugo kot začetno besedo bloka, ga s tem ne ohranja živega - notranji kazalci torej ne zagotavljajo, da se blok ne bo počistil.
- Kazalci v kopici morajo biti shranjeni na poravnanih naslovih.
- Kazalci v globalnih (in statičnih) spremenljivkah niso dovoljeni oziroma se ne bodo upoštevali pri določanju živosti objekta.
- Deluje le za enonitne programe.

### 4.2.1 Primer uporabe

Slika 4.3 prikazuje primer programa, ki uporablja čistilca pomnilnika. Najprej se ga inicializira s klicem `gc_init()`. Pomnilnik se dodeli s klici funkcije `gc_malloc(size_t size)` in kar je ključno, izvorna koda ne vsebuje ukazov za eksplicitno sproščanje pomnilnika. Zaradi definicije simbola `LOGGING` se ob čiščenju za vsak sproščen blok pomnilnika izpiše sporočilo z naslovom v konzolo. Za lažjo predstavo delovanja čistilca se na treh različnih mestih ročno proži čiščenje pomnilnika. Izpis konzole je prikazan na sliki 4.2.

```
First call to gc_collect() finished.

GC Log: Freeing unmarked memory block: 0x100200000
GC Log: Freeing unmarked memory block: 0x100200070
GC Log: Freeing unmarked memory block: 0x1002000e0
GC Log: Freeing unmarked memory block: 0x100200150
GC Log: Freeing unmarked memory block: 0x1002001c0
GC Log: Freeing unmarked memory block: 0x100200230
GC Log: Freeing unmarked memory block: 0x1002002a0
GC Log: Freeing unmarked memory block: 0x100200310
GC Log: Freeing unmarked memory block: 0x100200380
GC Log: Freeing unmarked memory block: 0x1002003f0
Second call to gc_collect() finished.

GC Log: Freeing unmarked memory block: 0x100200460
Third call to gc_collect() finished.

Program ended with exit code: 0
```

Slika 4.2: Izpis programa iz slike 4.3 v konzoli.

```c
#include <stdio.h>
#include <stdlib.h>
#include "garbagecollector.h"

 int main(int argc, const char * argv[]) {
     gc_init();

     void **root_ptr = gc_malloc(100);
     void **ptr = root_ptr;
     for (int i = 0; i < 10; i++) {
         ptr[i] = gc_malloc(100);
         ptr = ptr[i];
     }

     gc_collect();
     printf("First call to gc_collect() finished.\n\n");

     root_ptr = NULL;

     gc_collect();
     printf("Second call to gc_collect() finished.\n\n");

     ptr = NULL;

     gc_collect();
     printf("Third call to gc_collect() finished.\n\n");

     exit(0);
}
```

Slika 4.3: Primer programa z uporabo čistilca pomnilnika.

## 4.3 Implementacija čistilca

Implementacija čistilca zajema dodeljevanje in čiščenje pomnilnika. Izvorna koda je dostopna na [23].

### 4.3.1 Dodeljevanje pomnilnika

Implementacija funkcije `malloc` iz sistemske knjižnice v okolju Mac OS X dodeljuje pomnilnik iz t.i. dodelitvenega področja (*malloc zone*). Področje vsebuje spremenljivo število strani navideznega pomnilnika (velikosti 4 KB) iz katerih črpa pomnilniški prostor za dodelitve. Privzeto področje se ustvari ob prvem klicu funkcije `malloc` ali `calloc`. API knjižnice omogoča tudi ustvarjanje dodatnih področij. Za souporabo eksplicitnega in avtomatskega čiščenja pomnilnika smo vse dodelitve pomnilnika, ki jih naredi čistilec, preusmerili v novo dodelitveno področje.

Način dodeljevanja pomnilnika v posameznem dodelitvenem področju je odvisen od velikosti zahtevanega bloka:

- **Drobno dodeljevanje** (*tiny allocation*): do blokov velikosti 496 B za 32-bitne sisteme oziroma do velikosti 1008 B za 64-bitne sisteme. Pomnilnik se dodeli iz drobne regije velikosti 1 MB, ki vodi evidenco zasedenosti blokov velikosti 16 B (velikosti dodelitev se zaokrožijo na zgornjo vrednost večkratnika števila 16).

- **Majhno dodeljevanje** (*small allocation*): od blokov velikosti 497 B za 32-bitne sisteme oziroma od velikosti 1009 B za 64-bitne sisteme, do velikosti 127 KB. Pomnilnik se dodeli iz majhne regije velikosti 8 MB, ki vodi evidenco zasedenosti blokov velikosti 512 B (velikosti dodelitev se zaokrožijo na zgornjo vrednost večkratnika števila 512).

- **Veliko dodeljevanje** (*large allocation*): za bloke velikosti večje od 127 KB. Pomnilnik se dodeli neposredno kot strani navideznega pomnilnika (velikosti dodelitev se zaokrožijo na zgornjo vrednost večkratnika števila 4096).

Da bi zmanjšali pomnilniški prostor, ki ga potrebuje čistilec ob čiščenju, se že ob dodeljevanju pomnilnika ustvari bitno polje za vsako novo regijo oziroma vsako veliko dodeljevanje (sliki 4.4 in 4.5).

```
/*
 * Allocates requested memory and creates a new bitmap if new
 * region is created. Returns NULL on error.
 */
static void *gc_internal_alloc(size_t size, alloc_type_t type) {
    void *ptr = NULL;
    if (type == malloc_type)
        ptr = malloc_zone_malloc(zone, size);
    else if (type == calloc_type)
        ptr = malloc_zone_calloc(zone, 1, size);

    if (!ptr)                       // heap full
        return NULL;
    // check if malloc returned a new region
    // tiny allocation
    if (size <= SMALL_THRESHOLD) {
        // checks if it could be a new tiny region
        if (((intptr_t)ptr & TINY_BLOCK_MASK) == 0) {
            bitmap *tiny_bitmap = create_bitmap(ptr, TINY_BITMAP_SIZE);
            if (!tiny_bitmap) {     // allocation of bitmap failed
                free(ptr);          // free memory and return
                return NULL;
            }
        }
    } ...
```

Slika 4.4: Dodeljevanje pomnilnika.

### 4.3.2 Čiščenje pomnilnika

Izziv pri implementaciji konzervativnega čistilca predstavlja dobra hitrost izvajanja. Ker označevalna faza dominira celoten čas izvajanja čistilca, je

```
    ...
    // small allocation
    else if (size <= LARGE_THRESHOLD) {
        // checks if it could be a new small region
        if (((intptr_t)ptr & SMALL_BLOCK_MASK) == 0) {
            bitmap *small_bitmap = create_bitmap(ptr, SMALL_BITMAP_SIZE);
            if (!small_bitmap) {    // allocation of bitmap failed
                free(ptr);          // free memory and return
                return NULL;
            }
        }
    }
    // large allocation
    else {
        // large allocations need only mark-bit
        bitmap *large_bitmap = create_bitmap(ptr, 1);
        if (!large_bitmap) {        // allocation of bitmap failed
            free(ptr);              // free memory and return
            return NULL;
        }
    }
    return ptr;
}
```

Slika 4.5: Dodeljevanje pomnilnika (nad.).

potrebna še posebna pozornost pri učinkovitem prepoznavanju kazalcev. Implementacija `malloc` na operacijskem sistemu Mac OS X sledi vsem dodelitvam pomnilnika in izpostavlja funkcijo `malloc_zone_from_ptr(const void *ptr)`, ki za posredovan kazalec vrne področje, v kateri je podan blok pomnilnika dodeljen. Na ta način se lahko hitro preveri veljavnost kazalca.

Čiščenje poteka v dveh fazah:

1. **Označevanje**. Pregleda se vse korene mutatorja (vse splošno namenske registre in celoten sklad izvajalne niti) za veljavne kazalce (sliki 4.6 in 4.7). Če je le-ta najden, se nemudoma kliče funkcija `mark()`, ki glo-

binsko preišče del grafa objektov in ga sproti označuje (slika 4.8). Mac OS X ABI zagotavlja, da so kazalci na skladu poravnani [2]. Podobno omejimo tudi programski jezik C na poravnano shranjevanje kazalcev v kopici. Kazalce je zato potrebno preverjati le na pomnilniških lokacijah deljivih z 8 (na 64-bitnih sistemih), oziroma s 4 (na 32-bitnih sistemih), s čimer se občutno pohitri fazo označevanja.

2. **Pometanje**. Implementacija področnega dodeljevanja pomnilnika razkriva funkcijo `enumerator`, ki našteje vse dodeljene bloke pomnilnika v določenem področju. Kot argument prejme funkcijo povratnih klicev (*callback function*), ki jo proži ob obisku vsake regije. Faza pometanja (slika 4.9) jo najprej pokliče za sproščanje blokov pomnilnika, ki niso bili označeni v prejšnji fazi (slika 4.10), naslednji klic funkcije pa prestavi vsa bitna polja aktivnih regij v drugo zbirko (slika 4.11), da se lahko neuporabljena sprostijo.

```
/*
 * Marks all objects directly referenced by mutator's roots.
 */
static void mark_from_roots() {
    void **candidate; // needs checking if it can be a pointer
    // look for pointers in registers
    intptr_t *registers_i = registers;
    // loads all registers' values into memory
    __asm__ __volatile__ ("movq %%rax, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rbx, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rcx, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rdx, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rsp, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rbp, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rsi, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%rdi, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r8, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r9, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r10, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r11, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r12, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r13, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r14, %0" : "=r" (*registers_i++));
    __asm__ __volatile__ ("movq %%r15, %0" : "=r" (*registers_i++));
    for (candidate = (void *)registers; candidate < (void **)registers_i;
         candidate++) {
        if (valid_ptr(*candidate) && !is_marked(*candidate)) {
            set_marked(*candidate);
            // add memory block to worklist
            ptr_element *element = malloc(sizeof(*element));
            if (!element) {
                perror("GC: Not enough memory left for collection.");
                exit(1);
            }
            element->block = *candidate;
            LL_PREPEND(worklist, element);
            mark(); // immediately call mark() to reduce worklist size
        }
    } ...
```

Slika 4.6: Označevanje korenov.

```
    ...
    // look for pointers in the thread stack (stack grows downwards)
    void *stack_base_address = pthread_get_stackaddr_np(pthread_self());
    void *stack_lower_bound = get_sp();
    for (candidate = stack_base_address;
         (void *)candidate > stack_lower_bound; candidate--) {
        if (valid_ptr(*candidate) && !is_marked(*candidate)) {
            set_marked(*candidate);
            // add memory block to worklist
            ptr_element *element = malloc(sizeof(*element));
            if (!element) {
                perror("GC: Not enough memory left for collection.");
                exit(1);
            }
            element->block = *candidate;
            LL_PREPEND(worklist, element);
            mark(); // immediately call mark() to reduce worklist size
        }
    }
}
```

Slika 4.7: Označevanje korenov (nad.).

```
/*
 * Marks objects with depth-first traversal of object graph.
 */
static void mark() {
    // iterate until worklist is empty (it contains only marked objects)
    while (worklist != NULL) {
        // pop element from worklist
        ptr_element *pop_element = worklist;
        void **block = worklist->block;
        LL_DELETE(worklist, worklist);
        free(pop_element);

        // mark and add children to worklist
        size_t block_size = malloc_size(block);
        // pointers are aligned on 8 byte boundary (on 64-bit systems)
        for (int i = 0; i < (block_size / sizeof(void *)); i++) {
            void *child = block[i];
            if (valid_ptr(child) && !is_marked(child)) {
                set_marked(child);
                // add child to worklist
                ptr_element *child_el = malloc(sizeof(*child_el));
                if (!child_el) {
                    perror("GC: Not enough memory left for collection.");
                    exit(1);
                }
                child_el->block = child;
                LL_PREPEND(worklist, child_el);
            }
        }
    }
}
```

Slika 4.8: Globinsko preiskovanje grafa objektov in označevanje.

```
/*
 * Sweeps garbage memory blocks and deletes unused bitmaps
 * of regions removed.
 */
static void sweep() {
    // enumerate through all memory regions in use and free
    // unmarked memory blocks
    zone->introspect->enumerator(mach_task_self(),
                                 NULL,
                                 MALLOC_PTR_IN_USE_RANGE_TYPE,
                                 (vm_address_t)zone,
                                 NULL,
                                 sweep_range_recorder);
    // after sweeping move bitmaps of memory regions still in use
    // from bitmaps hashtable to in_use_bitmaps
    zone->introspect->enumerator(mach_task_self(),
                                 NULL,
                                 MALLOC_PTR_REGION_RANGE_TYPE,
                                 (vm_address_t)zone,
                                 NULL,
                                 bitmap_free_range_recorder);
    // delete and free unused bitmaps
    bitmap *current_bitmap, *tmp;
    HASH_ITER(hh, bitmaps, current_bitmap, tmp) {
        #ifdef LOGGING
        printf("GC Log: Removing unused bitmap for region: %p\n",
               current_bitmap->address);
        #endif
        HASH_DEL(bitmaps, current_bitmap);
        free(current_bitmap->bitmap);
        free(current_bitmap);
    }
    // move in_use_bitmaps to bitmaps
    bitmaps = in_use_bitmaps;
    in_use_bitmaps = NULL;
}
```

Slika 4.9: Faza pometanja.

```
/*
 * Range recorder function used to iterate over specified region and
 * collect (sweep) garbage.
 * Must be used with MALLOC_PTR_IN_USE_RANGE_TYPE enumerator type mask.
 */
static void sweep_range_recorder(task_t task, void *context,
                                 unsigned type_mask, vm_range_t *ranges,
                                 unsigned range_count) {
    // get hashmap for range to iterate through
    void *base_address = get_base_address((void *) ranges->address);
    unsigned int bit_pos;
    bitmap *to_check = NULL;
    HASH_FIND_PTR(bitmaps, &base_address, to_check);
    if (!to_check) {
        perror("GC: Bitmap not found.");
        exit(2);
    }
    // iterate through all allocated memory blocks and free unmarked ones
    vm_range_t *r, *end;
    for (r = ranges, end = ranges + range_count; r < end; r++) {
        bit_pos = get_bit_pos_bitmap((void *) r->address);
        // if marked
        if (get_bit_bitmap(to_check->bitmap, bit_pos)) {
            clear_bit_bitmap(to_check->bitmap, bit_pos); // clear mark bit
        }
        // if NOT marked
        else {
            #ifdef LOGGING
            printf("GC Log: Freeing unmarked memory block: %p\n",
                   (void *)r->address);
            #endif
            free((void *) r->address); // free unmarked memory block
        }
    }
}
```

Slika 4.10: Sproščanje pomnilnika neoznačenih objektov.

```
/*
 * Moves bitmap of each region encountered.
 */
static void bitmap_free_range_recorder(task_t task, void *context,
                                       unsigned type_mask, vm_range_t *ranges,
                                       unsigned range_count) {
    if (setjmp(buf)) // if HASH_ADD_PTR fails at allocating memory
        return;
    // for each range encountered, copy its bitmap to in_use_bitmaps
    bitmap *to_add;
    vm_range_t *r, *end;
    for (r = ranges, end = ranges + range_count; r < end; r++) {
        HASH_FIND_PTR(bitmaps, &(r->address), to_add);
        HASH_DEL(bitmaps, to_add);
        HASH_ADD_PTR(in_use_bitmaps, address, to_add);
    }
}
```

Slika 4.11: Prestavljanje aktivnih bitnih polj v drugo zbirko.

## 4.4 Primerjava programa s čistilcem in brez

Za vrednotenje sprememb v hitrosti izvajanja programa z uvedbo čistilca smo za primerjavo vzeli program, ki implementira urejen povezani seznam (sliki 4.12 in 4.13). Testni primer obsega dodajanje in brisanje 100.000 elementov v seznamu (slika 4.14).

Predvsem nas je zanimala hitrost izvajanja programa pred in po uvedbi avtomatskega čiščenja pomnilnika. Povprečen čas, ki ga je procesor porabil za izvajanje s čistilcem je znašal 20,78 s, brez čistilca pa 20,4 s (povprečje treh neodvisnih izvajanj). Uvedba čistilca je torej podaljšala čas izvajanja za približno 1,9%, kar je v večini primerov uporabe popolnoma sprejemljivo.

Zavedati pa se moramo, da merjenje na takšnih sintetičnih testih najpogosteje ne odraža dejanskega stanja v praksi. Eksperimentalno vrednotenje hitrosti čistilcev pomnilnikov je v splošnem težavno, saj so zaradi kaotičnega načina delovanja lahko rezultati že pri majhnih spremembah vhodnih parametrov zelo veliki. Rigoroznejše raziskave so sicer pokazale, da je ob dovolj veliki kopici hitrost izvajanja programa s čistilcem primerljiva z eksplicitnim upravljanjem pomnilnika, a je v povprečju za 17% slabša [11].

Skrita vrednost čistilca pomnilnika, ki se jo težko izrazi s številkami, je pa vsekakor vidna v izvorni kodi programa. Z odstranitvijo ukazov za eksplicitno sproščanje pomnilnika se težave z upravljanjem pomnilnika bistveno zmanjšajo, če ne celo odstranijo. Kako pomembno je to nakazuje tudi dejstvo, da praktično vsi programski jeziki danes delujejo s čistilcem pomnilnika.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "garbagecollector.h"
typedef struct node_ {
    int number;
    struct node_ *next;
} node;
void dodajU(int element, node **root) {
    node *iterator = *root;
    node *tempNext = NULL;
    if (iterator != NULL) { // check if the list is not empty
        if (element <= (*root)->number) {
            tempNext = (*root);
            if ((*root = gc_calloc(1, sizeof(**root))) != NULL) {
                (*root)->number = element;
                (*root)->next = tempNext;
            }
            else *root = tempNext;
            return;
        }
        while (iterator->next != NULL) {
            if (element <= iterator->next->number) break;
            iterator = iterator->next;
        }
        tempNext = iterator->next; // remember the pointer to the next
        if ((iterator->next = gc_calloc(1, sizeof(*iterator->next)))) {
            iterator = iterator->next;
            iterator->number = element;
            iterator->next = tempNext;
        }
        else iterator->next = tempNext;
        return;
    }
    else { // if list is empty, create the first node
        if ((*root = gc_calloc(1, sizeof(**root))) != NULL)
            (*root)->number = element;
        return;
    }
}
```

Slika 4.12: Implementacija urejenga povezanega seznama.

```
void brisi(int element, node **root) {
    node *iterator = *root;
    node *tempPrevious = NULL;
    node *tempNext = NULL;
    while (iterator != NULL) {
        if (element == iterator->number) {
            tempNext = iterator->next;
            if (iterator == *root) // if current node is the root
                *root = tempNext;
            else tempPrevious->next = tempNext;
            return;
        }
        else if (element < iterator->number) { // returns if no left
            return;
        }
        else {
            tempPrevious = iterator;
            iterator = iterator->next;
        }
    }
}
void izpisi(node **root)
{
    node *iterator = *root;
    printf("List:");
    while (iterator != NULL) {
        printf(" %d", iterator->number);
        iterator = iterator->next;
    }
    printf("\n");
}
```

Slika 4.13: Implementacija urejenga povezanega seznama (nad).

```
int main(int argc, const char * argv[])
{
    gc_init();
    node *root = NULL; // defines the root node
    clock_t begin, end;
    double time_spent;
    begin = clock();
    for (int i = 0; i < 100000; i++) {
        dodajU(i, &root);
    }
    for (int i = 0; i < 100000; i++) {
        brisi(i, &root);
    }
    gc_collect();
    end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("CPU time spent: %f\n", time_spent);
    exit(0);
}
```

Slika 4.14: Testni primer, ki doda in odstrani 100.000 elementov v povezanem seznamu.

# Poglavje 5

# Zaključek

Diplomska naloga nudi celovit pregled nad osnovnimi metodami čiščenja pomnilnika. Ključen del predstavlja lastna implementacija avtomatskega čiščenja pomnilnika.

Pogosto se ob uporabi programskega jezika sploh ne zavedamo zalednega delovanja, ki ga opravlja čistilec pomnilnika. V okoljih, kjer je hitrost delovanja programa bistvenega pomena, ima lahko pravilna izbira metode čiščenja pomnilnika velik vpliv na zmogljivost. Poznavanje lastnosti čistilcev nas pomaga usmeriti k primerni rešitvi za dane pogoje.

Uporaba čistilca za programski jezik C lahko močno poenostavi programiranje in v večji meri tudi odpravi napake, ki se pojavijo z eksplicitnim upravljanjem pomnilnika. Če hitrost ne predstavlja edinega vodila pri razvoju programske opreme, je uporaba čistilca vsekakor vredna premisleka.

Delovanje čistilca je zaenkrat omejeno na računalniško okolje Mac OS X (x86-64) zaradi tesnega sodelovanja čistilca s konkretno implementacijo družine funkcij `malloc` na tem sistemu ter zaradi eksplicitnega naštevanja registrov x86-64 arhitekture. S prilagoditvijo implementacije bi se dalo podpreti še druge platforme. Podobno smo morali omejiti tudi izraznost programskega jezika C zaradi hitrostnih pomislekov čistilca. Poravnanost kazalcev v kopici in neveljavnost notranjih kazalcev bi se dalo zaobiti s posledično nekoliko nižjo zmogljivostjo čistilca.

Velja omeniti še, da bi se za resnejšo analizo čistilca moralo opraviti več testiranj z različnimi programi in pridobljene rezultate primerjati tudi z drugimi čistilci.

# Literatura


[1] A garbage collector for C and C++. `http://www.hboehm.info/gc/`. [Dostopano dne 25. 8. 2014].

[2] System V application binary interface AMD64 architecture processor supplement. `http://people.freebsd.org/~obrien/amd64-elf-abi.pdf`. [Dostopano dne 20. 8. 2014].

[3] uthash: a hash table for C structures. `http://troydhanson.github.io/uthash/`. [Dostopano dne 27. 8. 2014].

[4] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. *SIGPLAN Not.*, 39(10):224–236, October 2004.

[5] Apple. Memory management policy. `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html`, 2012. [Dostopano dne 12. 8. 2014].

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[7] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, September 1976.

[8] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[9] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[10] Barry Hayes. Using key object opportunism to collect old objects. *SIGPLAN Not.*, 26(11):33–46, November 1991.

[11] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313–326, October 2005.

[12] Martin Hirzel, Amer Diwan, and Antony L. Hosking. On the usefulness of liveness for garbage collection and leak detection. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 181–206, London, UK, UK, 2001. Springer-Verlag.

[13] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[14] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[15] Richard E. Jones and Chris Ryder. A study of java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA, 2008. ACM.

[16] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[17] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[18] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society.

[19] E.J.H. Pepels, J. Plasmeijer, C.J.D. van Eekelen, and M.C.J.D. Eekelen. *A Cyclic Reference Counting Algorithm and Its Proof*. Internal report / Department of Informatics, Faculty of Science, University of Nijmengen. Department of Theoretical Computer Science and Computational Models, Faculty of Science, University of Nijmegen, 1988.

[20] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 101–112, New York, NY, USA, 2002. ACM.

[21] Tony Printezis and Alex Garthwaite. Visualising the train garbage collector. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 50–63, New York, NY, USA, 2002. ACM.

[22] Robert A. Saunders. The LISP system for the Q-32 computer. In *The Programming Language LISP: Its Operation and Applications*, pages 220–231. Information International, Inc., fourth edition, 1974.

[23] Andrej Česen. Garbage collector for C programming langauge. `https://github.com/MedusasPath/garbage-collector`, 2014. [Dostopano dne 15. 9. 2014].

[24] Paul R Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, pages 1–42. Springer, 1992.

[25] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 177–191, New York, NY, USA, 1991. ACM.